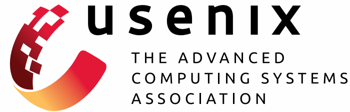


ReconFS: A Reconstructable File System on Flash Storage

Youyou Lu, Jiwu Shu, and Wei Wang, *Tsinghua University*

https://www.usenix.org/conference/fast14/technical-sessions/presentation/lu

This paper is included in the Proceedings of the
12th USENIX Conference on File and Storage Technologies (FAST '14).
February 17–20, 2014 • Santa Clara, CA USA
ISBN 978-1-931971-08-9

Open access to the Proceedings of the
12th USENIX Conference on File and Storage
Technologies (FAST '14)
is sponsored by

ReconFS: A Reconstructable File System on Flash Storage

Youyou Lu Jiwu Shu* Wei Wang
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
* *Corresponding author: shujw@tsinghua.edu.cn*
{*luyy09, wangwei11*}*@mails.tsinghua.edu.cn*

Abstract

Hierarchical namespaces (directory trees) in file systems are effective in indexing file system data. However, the update patterns of namespace metadata, such as intensive writeback and scattered small updates, exaggerate the writes to flash storage dramatically, which hurts both performance and endurance (i.e., limited program/erase cycles of flash memory) of the storage system.

In this paper, we propose a reconstructable file system, ReconFS, to reduce namespace metadata writeback size while providing hierarchical namespace access. ReconFS decouples the volatile and persistent directory tree maintenance. Hierarchical namespace access is emulated with the volatile directory tree, and the consistency and persistence of the persistent directory tree are provided using two mechanisms in case of system failures. First, consistency is ensured by embedding an inverted index in each page, eliminating the writes of the pointers (indexing for directory tree). Second, persistence is guaranteed by compacting and logging the scattered small updates to the metadata persistence log, so as to reduce write size. The inverted indices and logs are used respectively to reconstruct the structure and the content of the directory tree on reconstruction. Experiments show that ReconFS provides up to 46.3% performance improvement and 27.1% write reduction compared to ext2, a file system with low metadata overhead.

1 Introduction

In recent years, flash memory is gaining popularity in storage systems for its high performance, low power consumption and small size [11, 12, 13, 19, 23, 28]. However, flash memory has limited program/erase (P/E) cycles, and the reliability is weakened as P/E cycles approach the limit, which is known as the endurance problem [10, 14, 17, 23]. The recent trend of denser flash memory, which increases storage capacity by multiple-level cell (MLC) or triple-level cell (TLC) technologies, makes the endurance problem even worse [17].

File system design evolves slowly in the past few decades, yet it has a marked impact on I/O behaviors of the storage subsystems. Recent studies have proposed to revisit the namespace structure of file systems, e.g., flexible indexing for search-friendly file systems [33] and table structured metadata management for better metadata access performance [31]. Meanwhile, leveraging the internal storage management of flash translation layer (FTL) of solid state drives (SSDs) to improve storage management efficiency has also been discussed [19, 23, 25, 37]. But namespace management also impacts flash-based storage performance and endurance, especially when considering metadata-intensive workloads. This however has not been well researched.

Namespace metadata are intensively written back to persistent storage due to system consistency or persistence guarantees [18, 20]. Since the no-overwrite property of flash memory requires writes to be updated in free pages, frequent writeback introduces a large *dynamic update size* (i.e., the total write size of free pages that are used). Even worse, a single file system operation may scatter updates to different metadata pages (e.g., the create operation writes both the inode and the directory entry), and the average update size to each metadata page is far less than one page size (e.g., an inode in ext2 has the size of 128 bytes). A whole page needs to be written even though only a small part in the page is updated. Endurance, as well as performance, of flash storage systems is affected by namespace metadata accesses due to frequent and scattered small write patterns.

To address these problems, we propose a reconstructable file system, ReconFS, which provides a volatile hierarchical namespace and relaxes the writeback requirements. ReconFS decouples the maintenance of the volatile and persistent directory trees. Metadata

pages are written back to their home locations only when they are evicted or checkpointed (i.e., the operation to update the persistent directory tree the same as the volatile directory tree) from main memory. Consistency and persistence of the persistent directory tree are guaranteed using two new mechanisms. First, we use *embedded connectivity* mechanism to embed an inverted index in each page and track the unindexed pages. Since the namespace is tree-structured, the inverted indices are used for directory tree structure reconstruction. Second, we log the differential updates of each metadata page to the metadata persistence log and compact them into fewer pages, and we call it *metadata persistence logging* mechanism. These logs are used for directory tree content update on reconstruction.

Fortunately, flash memory properties can be leveraged to keep overhead of the two mechanisms low. First, page metadata, the spare space alongside each flash page, is used to store the inverted index. The inverted index is atomically accessed with its page data without extra overhead [10]. Second, unindexed pages are tracked in the unindexed zone by limiting new allocations to a continuous logical space. The address mapping table in FTL redirects the writes to different physical pages, and the performance is not affected even though the logical layout is changed. Third, high random read performance makes the compact logging possible, as the reads of corresponding base pages are fast during recovery. As such, ReconFS can efficiently gain performance and endurance benefits with rather low overhead.

Our contributions are summarized as follows:

- We propose a reconstructable file system design to avoid the high overhead of maintaining a persistent directory tree and emulate hierarchical namespace access using a volatile directory tree in memory.
- We provide namespace consistency by embedding an inverted index with the indexed data and eliminate the pointer update in the parent node (in the directory tree view) to reduce writeback frequency.
- We also provide metadata persistence by logging and compacting dirty parts from multiple metadata pages to the metadata persistence log, and the compact form reduces metadata writeback size.
- We implement ReconFS based on ext2 and evaluate it against different file systems, including ext2, ext3, btrfs and f2fs. Results show an up to 46.3% performance increase and 27.1% endurance improvement compared to ext2, a file system with low metadata overhead.

The rest of this paper is organized as follows. Section 2 gives the background of flash memory and namespace management. Section 3 describes the ReconFS design, including the decoupled volatile and persistent directory tree maintenance, the embedded connectivity and metadata persistence logging mechanisms, as well as the reconstruction. We present the implementation in Section 4 and evaluate ReconFS in Section 5. Related work is given in Section 6, and the conclusion is made in Section 7.

2 Background

2.1 Flash Memory Basics

Programming in flash memory is performed in one direction. Flash memory cells need to be erased before overwritten. The read/write unit is a flash page (e.g., 4KB), and the erase unit is a flash block (e.g., 64 pages). In each flash page, there is a spare area for storing the metadata of the page, which is called page metadata or out-of-band (OOB) area [10]. The page metadata is used to store error correction codes (ECC). And it has been proposed to expose the page metadata to software in NVMe standard [6].

Flash translation layers (FTLs) are used in flash-based solid state drives (SSDs) to export the block interface [10]. FTLs translate the logical page number in the software to the physical page number in flash memory. The address mapping hides the no-overwrite property from the system software. FTLs also perform garbage collection to reclaim space and wear leveling to extend the lifetime of the device.

Flash-based SSDs provide higher bandwidth and IOPS compared to hard disk drives (HDDs) [10]. Multiple chips are connected through multiple channels inside an SSD to provide internal parallelism, providing high aggregated bandwidth. Due to elimination of mechanical moving part, an SSD provides high IOPS. Endurance is another element that makes flash-based SSDs different from HDDs [10, 14, 17, 23]. Each flash memory cell has limited program/erase (P/E) cycles. As the P/E cycles approach the limit, the reliability of each cell drops dramatically. As such, endurance is a critical issue in system designs on flash-based storage.

2.2 Hierarchical Namespaces

Directory trees have been used in different file systems for over three decades to manage data in a hierarchical way. But hierarchical namespaces introduce high overhead to provide consistency and persistence for the directory tree. Also, static metadata organization amplifies the metadata write size.

Namespace Consistency and Persistence. Directories and files are indexed in a tree structure, the directory tree. Each page uses pointers to index its children in the directory tree. To keep the consistency of the directory

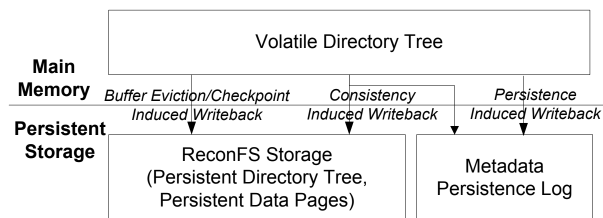


Figure 1: ReconFS Framework

tree, the page that has the pointer and the pointed page should be updated atomically. Different mechanisms, such as journaling [4, 7, 8, 34, 35] and copy-on-write (COW) [2, 32], are used to provide atomicity, but introduce a large amount of extra writes. In addition, the persistence requires the pointers to be in a durable state even after power failures, and this demands in-time writeback of these pages. This increases the writeback frequency, which also has a negative impact on endurance.

In this paper, we focus on the consistency of the directory tree, i.e., the metadata consistency. Data consistency can be achieved by incorporating transactional flash techniques [22, 23, 28, 29].

Metadata Organization. Namespace metadata are clustered and stored in the storage media, which we refer to as *static compacting*. Static compacting is commonly used in file systems. In ext2, index nodes in each block group are stored continuously. Since each index node is of 128 bytes in ext2, a 4KB page can store as many inodes as 32. Directory entries are organized in the similar way except that each directory entry is of variable length. Multiple directory entries with the same parent directory may share the same directory entry page. This kind of metadata organization improves the metadata performance in hard disk drives, as the metadata can be easily located.

Unfortunately, this kind of metadata organization has not addressed the endurance problem. For each file system operation, multiple metadata pages may be written but with only small parts updated in each page. E.g., a file create operation creates an inode in the inode page and writes a directory entry to the directory entry page. Since the flash-based storage is written in the unit of pages, the write amount is exaggerated by comparing the sum of all updated pages' size (from the view of storage device) with the updated metadata size (from the view of file system operations).

3 Design

ReconFS is designed to reduce the writes to flash storage while providing hierarchical namespace access. In this section, we first present the overall design of ReconFS, including the decoupled volatile and persistent directory tree maintenance and four types of metadata writeback. We then describe two mechanisms, *embedded connectivity* and *metadata persistence logging*, which provide consistency and persistence of the persistent directory tree with reduced writes, respectively. Finally, we discuss the ReconFS reconstruction.

3.1 Overview of ReconFS

ReconFS decouples the maintenance of the volatile and persistent directory trees. ReconFS emulates a volatile directory tree in main memory to provide the hierarchical namespace access. Metadata pages are updated to the volatile directory tree without being written back to the persistent directory tree. While the reduced writeback can benefit both performance and endurance of flash storage, consistency and persistence of the persistent directory tree need to be provided in case of unexpected system failures. Instead of writing back metadata pages directly to their home locations, ReconFS either embeds the inverted index with the indexed data for namespace consistency or compacts and writes back the scattered small updates in a log-structured way.

As shown in Figure 1, ReconFS is composed of three parts: the Volatile Directory Tree, the ReconFS Storage, and the Metadata Persistence Log. The Volatile Directory Tree manages namespace metadata pages in main memory to provide hierarchical namespace access. The ReconFS Storage is the persistent storage for ReconFS file system. It stores both the data and metadata, including the persistent directory tree, of the file system. The Metadata Persistence Log is a continuously allocated space in the persistent storage which is mainly used for the metadata persistence.

3.1.1 Decoupled Volatile and Persistent Directory Tree Maintenance

Since ReconFS emulates the hierarchical namespace access in main memory using a volatile directory tree, three issues are raised. First, namespace metadata pages need replacement when memory pressure is high. Second, namespace consistency is not guaranteed once system crashes without namespace metadata written back in time. Third, updates to the namespace metadata may get lost after unexpected system failures.

For the first issue, ReconFS writes back the namespace metadata to their home locations in ReconFS storage when they are evicted from the buffer, which we call *write-back on eviction*. This guarantees the metadata in persistent storage that do not have copies in main memory are the latest. Therefore, there are three kinds

of metadata in persistent storage (denoted as M_{disk}): the up-to-date metadata written back on eviction (denoted as $M_{up-to-date}$), the untouched metadata that have not been read into memory (denoted as $M_{untouched}$) and the obsolete metadata that have copies in memory (denoted as $M_{obsolete}$). Note $M_{obsolete}$ includes both pages that have dirty or clean copies in memory. Let M_{vdt}, M_{pdt} respectively be the namespace metadata of the volatile and persistent directory trees and M_{memory} be the volatile namespace metadata in main memory, we have

$$M_{vdt} = M_{memory} + M_{up-to-date} + M_{untouched},$$

$$M_{pdt} = M_{disk} = M_{obsolete} + M_{up-to-date} + M_{untouched}.$$

Since $M_{up-to-date}$ and M_{memory} are the latest, M_{vdt} is the latest. In contrast, M_{pdt} is not up-to-date, as ReconFS does not write back the metadata that still have copies in main memory. Volatile metadata are written back to their home locations for three cases: (1) file system unmount, (2) unindexed zone switch (Section 3.2), and (3) log truncation (Section 3.3). We call the operation that makes $M_{pdt} = M_{vdt}$ the *checkpoint* operation. When the volatile directory tree is checkpointed on unmount, it can be reconstructed by directly reading the persistent directory tree for later system booting.

The second and third issues are raised from unexpected system crashes, in which cases, $M_{vdt} \neq M_{pdt}$. The writeback of namespace metadata not only provides namespace connectivity for updated files or directories, but also keeps the descriptive metadata in metadata pages (e.g., owner, access control list in an inode) up-to-date. The second issue is caused by the loss of connectivity. To overcome this problem, ReconFS embeds an inverted index in each page for connectivity reconstruction (Section 3.2). The third issue is from the loss of metadata update. This problem is addressed by logging the metadata that need persistence (e.g., fsync) to the metadata persistence log (Section 3.3). In this way, the metadata of volatile directory tree can be reconstructed by first the connectivity reconstruction and then the descriptive metadata update even after system crashes.

3.1.2 Metadata Writeback

Metadata writeback to persistent storage, including the file system storage and the metadata persistence log, can be classified into four types as follows:

- *Buffer eviction induced writeback*: Metadata pages that are evicted due to memory pressure are written back to their home locations, so that these pages can be directly read out for later accesses without looking up the logs.
- *Checkpoint induced writeback*: Metadata pages are written back to their home locations for checkpoint operations, in order to reduce the reconstruction overhead.
- *Consistency induced writeback*: Writeback of pointers (used as the indices) is eliminated by embedding an inverted index with the indexed data of the flash storage, so as to reduce the writeback frequency.
- *Persistence induced writeback*: Metadata pages written back due to persistence requirements are compacted and logged to the metadata persistence log in a compact form to reduce the metadata writeback size.

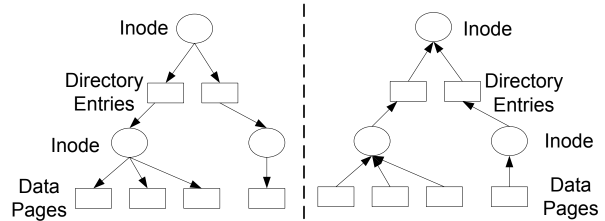


Figure 2: Normal Indexing (left) and Inverted Indexing (right) in a Directory Tree

3.2 Embedded Connectivity

Namespace consistency is one of the reasons why namespace metadata need frequent writeback to persistent storage. In the normal indexing of a directory tree as shown in the left half of Figure 2, the pointer and the pointed page of each link should be written back atomically for namespace consistency in each metadata operation. This not only requires the two pages to be updated but also demands journaling or ordered update for consistency. Instead, ReconFS provides namespace consistency using inverted indexing, which embeds the inverted index with the indexed data, as shown in the right half of Figure 2. Since the pointer is embedded with the pointed page, the consistency can be easily achieved. As well as the journal writes, the pointer updates are eliminated. In this way, the embedded connectivity lowers the frequency of metadata writeback and ensures the metadata consistency.

Embedded Inverted Index: In a directory tree, there are two kinds of links: links from directory entries to inodes (*dirent-inode links*) and links from inodes to data pages (*inode-data links*). Since directory entries are stored as data pages of directories in Unix/Linux, links from inodes to directory entries are classified as the inode-data links. For an inode-data link, the inverted index is the inode number and the data's location (i.e., the offset and length) in the file or directory. Since the inverted index is of several bytes, it is stored in the page

metadata of each flash page. For a dirent-inode link, the inverted index is the file or directory name and its inode number. Because the name is of variable length and is difficult to fit into the page metadata, an operation record, which is composed of the inverted index, the inode content and the operation type, is generated and stored in the metadata persistence log. The operation type in the operation record is set to 'creation' for create operations and 'link' for hard link operations. During reconstruction, the 'link' type does not invalidate previous creation records, while the 'creation' does.

An inverted index is also associated with a version number for identifying the correct version in case of inode number or directory entry reuses. When an inode number or a directory entry is reused after it is deleted, pages that belong to the deleted file or directory may still reside in persistent storage with their inverted indices. During reconstruction, these pages may be wrongly regarded as valid. To avoid this ambiguity, each directory entry is extended with a version number, and each inode is extended with the version pair $< V_{born}, V_{cur} >$, which indicates the liveness of the inode. V_{born} is the version number when the inode is created or reused. For a delete operation, V_{born} is set by increasing one to V_{cur}. Because all pages at that time have version numbers no larger than V_{cur}, all data pages of the deleted inode are set invalid. As same as the create and hard link operations, a delete operation generates a deletion record and appends it to the metadata persistence log, which is used to disconnect the inode from the directory tree and invalid all its children pages.

Unindexed Zone: Pages whose indices have not been written back are not accessible in the directory tree after system failures. These pages are called *unindexed pages* and need to be tracked for reconstruction. ReconFS divides the logical space into several zones and restricts the writes to one zone in each stage. This zone is called the *unindexed zone*, and it tracks all unindexed pages at one stage. A stage is the time period when the unindexed zone is used for allocation. When the zone is used up, the unindexed zone is switched to another. Before the zone switch, a checkpoint operation is performed to write the dirty indices back to their home locations. The restriction of writes to the unindexed zone incurs little performance penalty. This is because the FTL inside an SSD remaps logical addresses to physical addresses, and data layout in the logical space view does little impact on system performance while data layout in the physical space view is critical.

In addition to namespace connectivity, bitmap write-back is another source of frequent metadata persistence. The bitmap updates are frequently written back to keep the space allocation consistent. ReconFS only keeps the volatile bitmap in main memory, which is used for logical space allocation, and does not keep the persistent bitmap up-to-date. Once system crashes, bitmaps are reconstructed. Since new allocations are performed only in the unindexed zone, the bitmap in the unindexed zone is reconstructed using the valid and invalid statuses of the pages. Bitmaps in other zones are only updated when pages are deleted, and these updates can be reconstructed using deletion records in the metadata persistence log.

3.3 Metadata Persistence Logging

Metadata persistence causes frequent metadata write-back. The scattered small update pattern of the writeback amplifies the metadata writes, which are written back in the unit of pages. Instead of using static compacting (as mentioned in Section 2), ReconFS dynamically compacts the metadata updates and writes them to the metadata persistence log. While static compacting requires the metadata updates written back to their home locations, dynamic compacting is able to cluster the small updates in a compact form. Dynamic compacting only writes the dirty parts rather than the whole pages, so as to reduce write size.

In metadata persistence logging, writeback is triggered when persistence is needed, e.g., explicit synchronization or the wake up of pdflush daemon. The metadata persistence logging mechanism keeps track of the dirty parts of each metadata page in main memory and compacts those parts into the logs:

- Memory Dirty Tagging: For each metadata operation, metadata pages are first updated in the main memory. ReconFS records the location metadata (i.e., the offset and the length) of the dirty parts in each updated metadata page. The location metadata are attached to the buffer head of the metadata page to track the dirty parts for each page.
- Writeback Compacting: During writeback, ReconFS travels multiple metadata pages and appends their dirty parts to the log pages. Each dirty part has its location metadata (i.e., the base page address, the offset and length in the page) attached in the head of each log page.

Log truncation is needed when the metadata persistence log runs short of space. Instead of merging the small updates in the log with base metadata pages, ReconFS performs a checkpoint operation to write back all dirty metadata pages to their home locations. To mitigate the writeback cost, the checkpoint operation is performed in an asynchronous way using a writeback daemon, and the daemon starts when the log space drops below a pre-defined threshold. As such, the log is truncated without costly merging operations.

Multi-page Update Atomicity. Multi-page update atomicity is needed for an operation record which size

is larger than one page (e.g., a file creation operation with a 4KB file name). To provide the consistency of the metadata operation, these pages need to be updated atomically. Single-page update atomicity is guaranteed in flash storage, because the no-overwrite property of flash memory requires the page to be updated in a new place followed by atomic mapping entry update in the FTL mapping table.

Multi-page update atomicity is simply achieved using a flag bit in each page. Since a metadata operation record is written in continuously allocated log pages, the atomicity is achieved by tagging the start and end of these pages. The last page is tagged with flag '1', and the others are tagged with '0'. The bit is stored in the head of each log page. It is set when the log page is written back, and it does not require extra writes. During recovery, the flag bit '1' is used to determine the atomicity. Pages between two '1's belong to complete operations, while pages at the log tail without an ending '1' belong to an incomplete operation. In this way, multi-page update atomicity is achieved.

3.4 ReconFS Reconstruction

During normal shutdowns, the volatile directory tree writes the checkpoint to the persistent directory tree in persistent storage, which is simply read into main memory to reconstruct the volatile directory tree for the next system start. But once the system crashes, ReconFS needs to reconstruct the volatile directory tree using the metadata recorded by the embedded connectivity and the metadata persistence logging mechanisms. Since the persistent directory tree is the checkpoint of volatile directory tree when the unindexed zone is switched or the log is truncated, all page allocations are performed in the unindexed zone, and all metadata changes have been logged to the persistent metadata logs. Therefore, ReconFS only needs to update the directory tree by scanning the unindexed zone and the metadata persistence log. ReconFS reconstruction includes:

1. *File/directory reconstruction*: Each page in the unindexed zone is connected to its index node using its inverted index. And then, each page checks the version number in its inverted index with the $< V_{born}, V_{cur} >$ in its index node. If this matches, the page is indexed to the file or directory. Otherwise, the page is discarded because the page has been invalidated. After this, all pages, including file data pages and directory entry pages, are indexed to their index nodes.
2. *Directory tree connectivity reconstruction*: The metadata persistence log is scanned to search the dirent-inode links. These links are used to connect those inodes to the directory tree, so as to update the directory tree structure.
3. *Directory tree content update*: Log records in the metadata persistence log are used to update the metadata pages in the directory tree, so the content of the directory tree is updated to the latest.
4. *Bitmap reconstruction*: The bitmap in the unindexed zone is reset by checking the valid status of each page, which can be identified using version numbers. Bitmaps in other zones are not changed except for deleted pages. With the deletion or truncation log records, the bitmaps are updated.

After the reconstruction, those obsolete metadata pages in persistent directory tree are updated to the latest, and the recent allocated pages are indexed into the directory tree. The volatile directory tree is reconstructed to provide hierarchical namespace access.

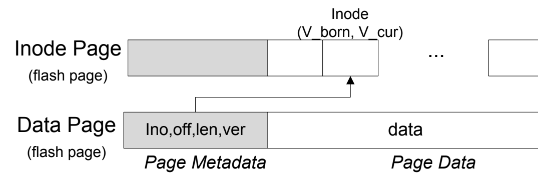


Figure 3: An Inverted Index for an Inode-Data Link

4 Implementation

ReconFS is implemented based on ext2 file system in Linux kernel 3.10.11. ReconFS shares both on-disk and in-memory data structures of ext2 but modifies the namespace metadata writeback flows.

In volatile directory directory tree, ReconFS employs two dirty flags for each metadata buffer page: persistence dirty and checkpoint dirty. *Persistence dirty* is tagged for the writeback to the metadata persistence log. *Checkpoint dirty* is tagged for the writeback to the persistent directory tree. Both of them are set when the buffer page is updated. The persistence dirty flag is cleared only when the metadata page is written to the metadata persistence log for metadata persistence. The checkpoint dirty flag is cleared only when the metadata are written back to its home location. ReconFS uses the double dirty flags to separate metadata persistence (the metadata persistence log) from metadata organization (the persistent directory tree).

In embedded connectivity, inverted indices for inode-data and dirent-inode links are stored in different ways. The inverted index of an *inode-data link* is stored in the page metadata of each flash page. It has the form of (ino, off, len, ver), in which *ino* is the inode number, off and len are the offset and the valid data length in the file or directory, respectively, and *ver* is the version number of the inode. The inverted index of a *dirent-*

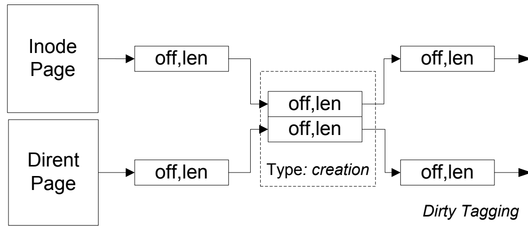


Figure 4: Dirty Tagging in Main Memory

inode link is stored as a log record with the record type *type* set to ‘creation’ in the metadata persistence log. The log record contains both the directory entry and the inode content and keeps an (off, len, lba, ver) extent for each of them. lba is the logical block address of the base metadata page. The log record acts as the inverted index for the inode, which is used to reconnect it to the directory tree. Unindexed zone in ReconFS is set by clustering multiple block groups in ext2. ReconFS limits the new allocations to these block groups, thus making these block groups as the unindexed zone. The addresses of these block groups are kept in file system super block and are made persistent on each zone switch.

In metadata persistence logging, ReconFS tags the dirty parts of each metadata page using a linked list, as shown in Figure 4. Each node in the linked list is a pair of (off, len) to indicate which part is dirty. Before each insertion, the list is checked to merge the overlapped dirty parts. The persistent log record also associates the type $type$, the version number ver and the logical block address lba for each metadata page with the linked list pairs, followed by the dirty content. In current implementation, ReconFS writes the metadata persistence log as a file in the root file system. Checkpoint is performed for file system unmount, unindexed zone switch or log truncation. Checkpoint for file system unmount is performed when the unmount command is issued, while checkpoint for the other two is triggered when the free space in the unindexed zone or the metadata persistence log drops below 5%.

Reconstruction of ReconFS is performed in three phases:

1. Scan Phase: Page metadata from all flash pages in the unindexed zone and log records from the metadata persistence log are read into memory. After this, all addresses of the metadata pages that appear in either of them are collected. And then, all these metadata pages are read into memory.
2. Zone Processing Phase: In the unindexed zone, each flash page is connected to its inode using the inverted index in its page metadata. Structures of files and directories are reconstructed, but they may have obsolete pages.
3. Log Processing Phase: Each log record is used either to connect a file or directory to the directory tree or to update the metadata page content. For a creation or hard link log record, the directory entry is updated for the inode. For a deletion or truncation log record, the corresponding bitmaps are read and updated. The other log records are used to update the page content. And finally, versions in the pages and inodes are checked to discard the obsolete pages, files and directories.

Table 1: File Systems

ext2	a traditional file system without journaling
ext3	a traditional journaling file system (journaled version of ext2)
btrfs[2]	a recent copy-on-write (COW) file system
f2fs[12]	a recent log-structured file system optimized for flash

5 Evaluation

We evaluate the performance and endurance of ReconFS against previous file systems, including ext2, ext3, btrfs and F2FS, and aim to answer the following four questions:

1. How does ReconFS compare with previous file systems in terms of performance and endurance?
2. What kind of operations gain more benefits from ReconFS? What are the benefits from embedded connectivity and metadata persistence logging?
3. What is the impact of changes in memory size?
4. What is the overhead of checkpoint and reconstruction in ReconFS?

In this section, we first describe the experimental setup before answering the above questions.

5.1 Experimental Setup

We implement ReconFS in Linux kernel 3.10.11, and evaluate the performance and endurance of ReconFS against the file systems listed in Table 1.

We use four workloads from filebench benchmark [3]. They emulate different types of servers. Operations and read-write ratio [21] of each workload are illustrated as follows:

- *fileserver* emulates a file server, which performs a sequence of create, delete, append, read, write and attribute operations. The read-write ratio is 1:2.
- *webproxy* emulates a web proxy server, which performs a mix of create-write-close, open-read-close and delete operations, as well as log appends. The read-write ratio is 5:1.

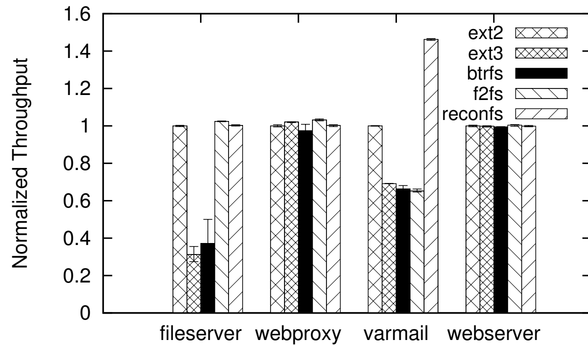


Figure 5: System Comparison on Performance

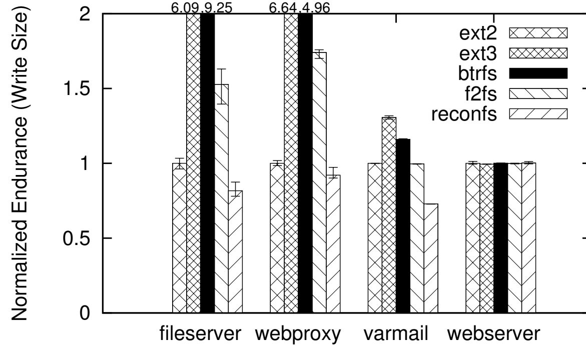


Figure 6: System Comparison on Endurance

Table 2: SSD Specification

Capacity	128 GB
Seq. Read Bandwidth	260 MB/s
Seq. Write Bandwidth	200 MB/s
Rand. Read IOPS (4KB)	17,000
Rand. Write IOPS (4KB)	5,000

- *varmail* emulates a mail server, which performs a set of create-append-sync, read-append-sync, read and delete operations. The read-write ratio is 1:1.
- *webserver* emulates a web server, which performs open-read-close operations, as well as log appends. The read-write ratio is 10:1.

Experiments are carried out on Fedora 10 using Linux kernel 3.10.11, and the computer is equipped with 4-core 2.50GHz processor and 12GB memory. We evaluate all file systems on a 128GB SSD, and its specification is shown in Table 2. All file systems are mounted with default options.

5.2 System Comparison

5.2.1 Overall Comparison

We evaluate the performance of all file systems by measuring the throughput reported by the benchmark, and the endurance by measuring the write size to storage. The write size to storage is collected from the block level trace using blktrace tool [1].

Figure 5 shows the throughput normalized to the throughput of ext2 to evaluate the performance. As shown in the figure, ReconFS is among the best of all file systems for all evaluated workloads, and gains performance improvement up to 46.3% than ext2 for varmail, the metadata intensive workload. For read intensive workloads, such as webproxy and webserver, all evaluated file systems do not show a big difference. But for write intensive workloads, such as fileserver and varmail, they show different performance. Ext2 shows comparatively higher performance than other file systems excluding ReconFS. Both ext3 and btrfs have provided namespace consistency with different mechanisms, e.g., waiting until the data reach persistent storage before writing back the metadata, but with poorer performance compared to ext2. F2FS, the file system with data layout optimized for flash, shows a comparable performance to ext2, but has inferior performance in varmail workload, which is metadata intensive and has frequent fsyncs. Comparatively, ReconFS achieves the performance of ext2 in all evaluated workloads, nearly the best performance of all previous file systems, and is even better than ext2 in varmail workload. Moreover, ReconFS provides namespace consistency with embedded connectivity while ext2 does not.

Figure 6 shows the write size to storage normalized to that of ext2 to evaluate the endurance. From the figure, we can see ReconFS effectively reduces write size for metadata and reduces write size by up to 27.1% compared to ext2. As same as the performance, the endurance of ext2 is the best of all file systems excluding ReconFS. On the while, ext3, btrfs and F2FS uses journaling or copy-on-write to provide consistency, which introduces extra writes. For instance, btrfs has the write size 9 times as large as that of ext2 in the fileserver workload. ReconFS provides namespace consistency using embedded connectivity without incurring extra writes, and further reduces write size by compacting metadata writeback. As shown in the figure, ReconFS shows a write size reduction of 18.4%, 7.9% and 27.1% even compared with ext2 respectively for fileserver, webproxy and varmail workloads.

5.2.2 Performance

To understand the performance impact of ReconFS, we evaluate four different operations that have to update the index node page and/or directory entry page. The four operations are file creation, deletion, append and append with fsyncs. They are evaluated using micro-

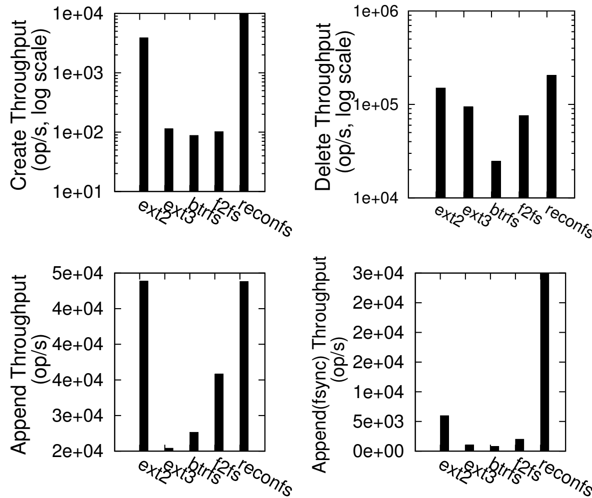


Figure 7: Performance Evaluation of Operations (File create, delete, append and append with fsync)

benchmarks. The file creation and deletion benchmarks create or delete 100K files spread over 100 directories. *fsync* is performed following each creation. The append benchmark appends 4KB pages to a file, and it inserts a fsync for every 1,000 (one fsync per 4MB) and 10 (one fsync per 40KB) append operations respectively for evaluating append and append with fsyncs.

Figure 7 shows the throughput of the four operations. ReconFS shows a significant throughput increase in file creation and append with fsyncs. File creation throughput in ReconFS doubles the throughput in ext2. This is because only one log page is appended in the metadata persistence log, while multiple pages need to be written back in ext2. Other file systems have even worse file creation performance due to consistency overheads. File deletion operations in ReconFS also show better performance than the others. File append throughput in ReconFS almost equals that in ext2 for append operations with one fsync per 1,000 append operations. But file append (with fsyncs) throughput in ext2 drops dramatically as the fsync frequency increases from 1/1000 to 1/10, as well as in the other journaling or log-structured file systems. In comparison, file append (with fsyncs) throughput in ReconFS only drops to half of previous throughput. When fsync frequency is 1/10, ReconFS has file append throughput 5 times better than ext2 and orders of magnitude better than the other file systems.

5.2.3 Endurance

To further investigate the endurance benefits of ReconFS, we measure the write size of ext2, ReconFS without log compacting (denoted as ReconFS-EC) and ReconFS.

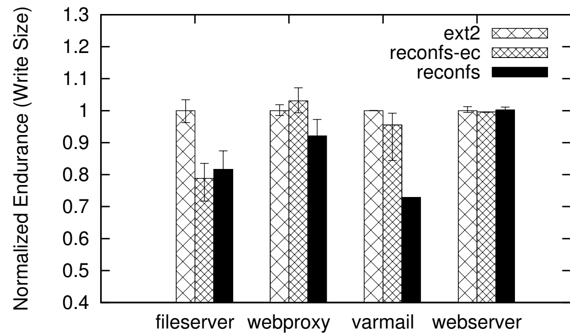


Figure 8: Endurance Evaluation for Embedded Connectivity and Metadata Persistence Logging

Figure 8 shows write sizes of the three file systems. We compare the write sizes of ext2 and ReconFS-EC to evaluate the benefit from embedded connectivity, since ReconFS-EC implements the embedded connectivity but without log compacting. From the figure, we observe that the fileserver workload shows a remarkable drop in write size from ext2 to ReconFS-EC. The benefit mainly comes from the intensive file creates and appends in the fileserver workload, which otherwise requires index pointers to be updated for namespace connectivity. Embedded connectivity in ReconFS eliminates updates to these index pointers. We also compare the write sizes of ReconFS-EC and ReconFS to evaluate the benefit from log compacting in metadata persistence logging. As shown in the figure, ReconFS shows a large write reduction in varmail workload. This is because frequent fsyncs reduce the effects of buffering, in other words, the updates to metadata pages are small when written back. As a result, the log compacting gains more improvement than other workloads.

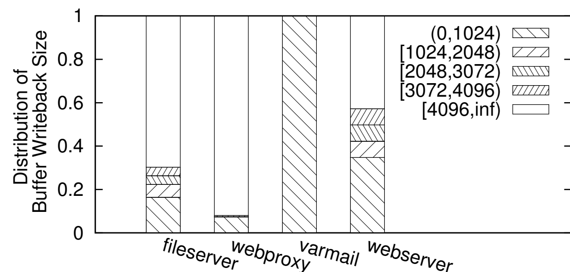
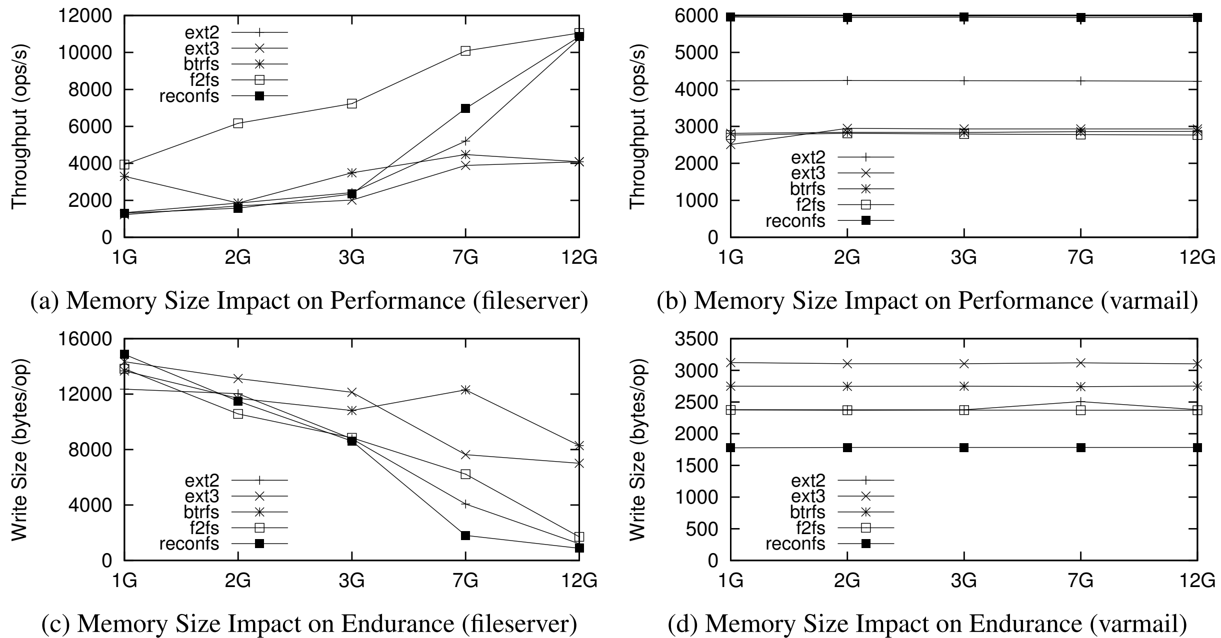


Figure 9: Distribution of Buffer Page Writeback Size

Figure 9 also shows the distribution of buffer page writeback size, which is the size of dirty parts in each page. As shown in the figure, over 99.9% of the dirty data for each page in metadata writeback of varmail workload are less than 1KB due to frequent fsyncs, while the others have the fraction varied from 7.3% to 34.7%



(a) Memory Size Impact on Performance (fileserver)

(b) Memory Size Impact on Performance (varmail)

(c) Memory Size Impact on Endurance (fileserver)

(d) Memory Size Impact on Endurance (varmail)

Figure 10: Memory Size Impact on Performance and Endurance

Table 3: Comparision of Full-Write and Compact-Write

Workloads	Full Write Size (KB)	Comp. Write Size (KB)	Compact Ratio
fileserver	108,143	48,624	44.96%
webproxy	45,133	21,325	47.25%
varmail	3,060,116	117,235	3.83%
webserver	374	143	38.36%

for dirty size less than 1KB. In addition, we calculate the compact ratio by dividing the full page update size with the compact write size, as shown in Table 3. The compact ratio of varmail workload achieves as low as 3.83%.

5.3 Impact of Memory Size

To study the memory size impact, we set the memory size to 1, 2, 3, 7 and 12 gigabytes[1] and measure both performance and endurance of all evaluated file systems. We measure performance in the unit of the operations per second (ops/s), and endurance in the unit of bytes per operation (bytes/op) by dividing the total write size with the number of operations. Results of webproxy and webserver workloads are not shown due to space limitation, as they are read intensive workloads and show little difference between file systems.

[1] We limit the memory size to 1, 2, 4, 8 and 12 gigabytes in the GRUB. The recognized memory sizes (shown in /proc/meminfo) are 997, 2,005, 3,012, 6,980 and 12,044 megabytes, respectively.

Figure 10 (a) shows the throughput of fileserver workload for all file systems under different memory sizes. As shown in the figure, ReconFS gains more when memory size becomes larger, in which case data pages are written back less frequently and the writeback of metadata pages has larger impact. When memory size is small and memory pressure is high, the impact of data writes dominates. ReconFS has poorer performance than F2FS, which has optimized data layout. When memory size increases, the impact from the metadata writes increases. Little improvement is gained in ext3 and btrfs when memory size increases from 7GB to 12GB. In contrast, ReconFS and ext2 gain significant improvement for their low metadata overhead and approach the performance of F2FS. Figure 10 (c) shows the endurance measured in bytes per operation of fileserver. In the figure, ReconFS has comparable or less write size than other file systems.

Figure 10 (b) shows the throughput of varmail workload. Performance is stable under different memory sizes, and ReconFS achieves the best performance. This is because varmail workload is metadata intensive workload and has frequent fsync operations. Figure 10 (d) shows the endurance of varmail workload. ReconFS achieves the best in all file systems.

5.4 Reconstruction Overhead

We measure the unmount time to evaluate the overhead of checkpoint, which writes back all dirty metadata to make the persistent directory tree equivalent to the

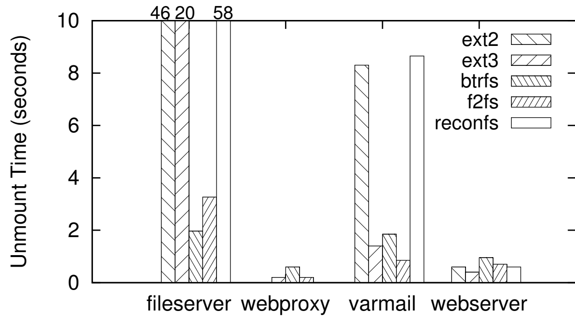


Figure 11: Unmount Time (Immediate Unmount)

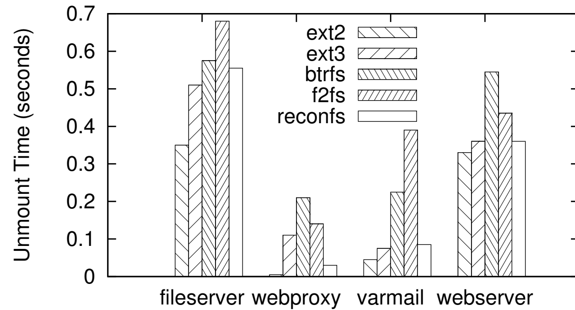


Figure 12: Unmount Time (Unmount after 90s)

volatile directory tree, as well as the reconstruction time.
Unmount Time. We use *time* command to measure the time of unmount operations and use the elapsed time reported by the time command.

Figure 11 shows the unmount time when the unmount is performed immediately when each benchmark completes. The read intensive workloads, webproxy and webserver, have unmount time less than one second for all file systems. But the write intensive workloads have various unmount time for different file systems. The unmount time in ext2 is 46 seconds, while that of ReconFS is 58. All the unmount time values are less than one minute, and they include the time used for both data and metadata writeback. Figure 12 shows the unmount time when the unmount is performed 90 seconds later after each benchmark completes. All of them are less than one second, and ReconFS does not show a noticeable difference with others.

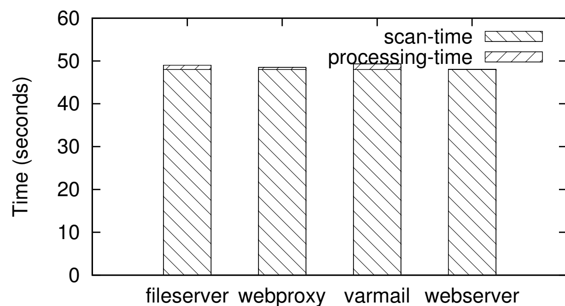


Figure 13: Recovery Time

Reconstruction Time. Reconstruction time has two main parts: scan time and processing time. The scan time includes the time of the unindexed zone scan and the log scan. The scan is the sequential read, which performance is bounded by the device bandwidth. The processing time is the time used to read the base metadata pages in the directory tree to be updated in addition to the recovery logic processing time. As shown in Figure 13, the scan time is 48 seconds for an 8GB zone on the SSD, and the processing time is around one second. The scan time is expected to be reduced with PCIe SSDs. E.g., the scan time for a 32GB zone on a PCIe SSD with 3GB/s is around ten seconds. Therefore, with high read bandwidth and IOPS, the reconstruction of ReconFS can complete in tens of seconds.

6 Related Work

File System Namespace. Research on file system namespace has been long for efficient and effective namespace metadata management. Relational database or table-based technologies have been used to manage namespace metadata for either consistency or performance. Inversion file system [26] manages namespace metadata using PostGRES database system to provide transaction protection and crash recovery to the metadata. TableFS [31] stores namespace metadata in LevelDB [5] to improve metadata access performance by leveraging the log-structured merge tree (LSM-tree) [27] implemented in LevelDB.

The hierarchical structure of namespace has also been discussed to be implemented in a flexible way to provide semantic accesses. Semantic file system [16] removes the tree-structured namespace and accesses files and directories using attributes. hFAD [33] proposes a similar approach, which prefers a search-friendly file system to a hierarchical file system.

Pilot [30] proposes an even aggressive way and eliminates all indexing in file systems, in which files are accessed only through a 64-bit universal identifier (UID). And Pilot does not provide tree-structured file access. Comparatively, ReconFS removes only the indexing of persistent storage to lower the metadata cost, and it emulates the tree-structured file access using the volatile directory tree.

Backpointers and Inverted Indices. Backpointers have been used in storage systems for different purposes. BackLog [24] uses backpointer in data blocks to reduce

the pointer updates when data blocks are moved due to advanced file system features, such as snapshots, clones. NoFS [15] uses backpointer for consistency checking on each read to provide consistency. Both of them use backpointer as the assistant to enhance new functions, but ReconFS uses backpointers (inverted indices) as the only indexing (without forward pointers).

In flash-based SSDs, backpointer (e.g., the logical page addresses) is stored in the page metadata of each flash page, which is atomically accessed with the page data, to recover the FTL mapping table [10]. On each device booting, all pages are scanned, and the FTL mapping table is recovered using the backpointer. OFSS [23] uses backpointer in page metadata in a similar way. OFSS uses an object-based FTL, and the backpointer in each page records the information of the object, which is used to delay the persistence of the object indexing. ReconFS extends the use of backpointer in flash storage to the file system namespace management. Instead of maintaining the indexing (forward pointers), ReconFS embeds only the reverse index (backward pointers) with the indexed data, and the reverse indices are used for reconstruction once system fails unexpectedly.

File System Logging. File systems have used logging in two different ways. One is the journaling, which updates metadata and/or data in the journaling area before updating them to their home locations, and is widely used in modern file systems to provide file system consistency [4, 7, 8, 34, 35]. Log-structured file systems use logging in the other way [32]. Log-structured file systems write all data and metadata in a logging way, making random writes sequential for better performance.

ReconFS employs the logging mechanism for metadata persistence. Unlike journaling file systems or log-structured file systems, which require tracking of valid and invalid pages for checkpoint and garbage cleaning, the metadata persistence log in ReconFS is simply discarded after the writeback of all volatile metadata. ReconFS also enables compact logging, because the base metadata pages can be read quickly during reconstruction due to high random read performance of flash storage.

File Systems on Flash-based Storage. In addition to embedded flash file systems [9, 36], researchers are proposing new general-purpose file systems for flash storage. DFS [19] is a file system that directly manages flash memory by leveraging functions (e.g., block allocation, atomic update) provided by FusionIO's ioDrive. Nameless Write [37] also removes the space allocation function in the file system and leverage the FTL space management for space allocation. OFSS [23] proposes to directly manage flash memory using an object-based FTL, in which the object indexing, free space management and data layout can be optimized with the flash memory characteristics. F2FS [12] is a promising log-structured file system which is designed for flash storage. It optimizes data layout in flash memory, e.g., the hot/cold data grouping. But these file systems have paid little attention to the high overhead of namespace metadata, which are frequently written back and are written in the scattered small write pattern. ReconFS is the first to address the namespace metadata problem on flash storage.

7 Conclusion

Properties of namespace metadata, such as intensive writeback and scattered small updates, make the overhead of namespace management high on flash storage in terms of both performance and endurance. ReconFS removes maintenance of the persistent directory tree and emulates hierarchical access using a volatile directory tree. ReconFS is reconstructable after unexpected system failures using both *embedded connectivity* and *metadata persistence logging* mechanisms. Embedded connectivity enables directory tree structure reconstruction by embedding the reverted index with the indexed data. With elimination of updates to parent pages (in the directory tree) for pointer updating, the consistency maintenance is simplified and the writeback frequency is reduced. Metadata persistence logging provides persistence to metadata pages, and the logged metadata are used for directory tree content reconstruction. Since only the dirty parts of metadata pages are logged and compacted in the logs, the writeback size is reduced. Reconstruction is fast due to high bandwidth and IOPS of flash storage. Through the new namespace management, ReconFS improves both performance and endurance of flash-based storage system without compromising consistency or persistence.

Acknowledgments

We would like to thank our shepherd Remzi Arpaci-Dusseau and the anonymous reviewers for their comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 60925006), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), Shanghai Key Laboratory of Scalable Computing and Systems, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, Huawei Technologies Co. Ltd., and Tsinghua University Initiative Scientific Research Program.

References

[1] blktrace(8) - linux man page. http://linux.die.net/man/8/blktrace.

[2] Btrfs. http://btrfs.wiki.kernel.org.

[3] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Main_Page.

[4] Journaled file system technology for linux. http://jfs.sourceforge.net/.

[5] LevelDB, a fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb/.

[6] The NVM express standard. http://www.nvmexpress.org.

[7] ReiserFS. http://reiser4.wiki.kernel.org.

[8] XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs/.

[9] Yaffs. http://www.yaffs.net.

[10] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX'08)*, 2008.

[11] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009.

[12] Neil Brown. An F2FS teardown. http://lwn.net/Articles/518988/.

[13] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, 2009.

[14] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, 2011.

[15] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[16] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)*, 1991.

[17] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[18] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[19] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: a file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[20] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[21] Eunji Lee, Hyokyung Bahn, and Sam H Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[22] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD'13)*, 2013.

[23] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[24] Peter Macko, Margo I Seltzer, and Keith A Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the*

8th USENIX Conference on File and storage technologies (FAST'10), 2010.

[25] David Nellans, Michael Zappe, Jens Axboe, and David Flynn. ptrim ()+ exists (): Exposing new FTL primitives to applications. In *the 2nd Annual Non-Volatile Memory Workshop*, 2011.

[26] Michael A Olson. The design and implementation of the inversion file system. In *USENIX Winter*, 1993.

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[28] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, 2011.

[29] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[30] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, 1980.

[31] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX'13)*, 2013.

[32] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[33] Margo I Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009.

[34] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, 2000.

[35] Stephen C Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[36] David Woodhouse. Jffs2: The journalling flash file system, version 2. `http://sourceware.org/jffs2`.

[37] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.